A Code Generation Metamodel for ULF-Ware

Generating Code for SDL and Interfacing with the Runtime Library

Michael Piefel and Toby Neumann

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
{piefel|tneumann}@informatik.hu-berlin.de

Abstract. Models can be used in many stages of many different processes, but in software engineering, the ultimate purpose of modelling is often code generation. While code can be generated from any model, we propose to use an intermediate model that is tailored to code generation instead. In order to be able to easily support different target languages, this model should be general enough; in order to support the whole process, the model has to contain behavioural as well as structural aspects. This paper introduces such a model and the ideas behind it.
When the model is to cover several languages, differences occur also in the available library functions. Furthermore, the input languages (e. g. SDL) may contain high-level concepts such as signal routing that are not easily mapped into simple instructions. We propose a runtime library to address both challenges.

1 Introduction

Models, in general, do not relate to programs at all. However, in software engineering models do refer to systems and their components, which may be executable code. Ideally, model-driven development (such as embodied in the OMG's Model Driven Architecture (MDA)) eventually leads to code generation.

Our research group has been involved in simulation and modelling for a long time. We developed SITE [1], a compiler and runtime environment for the Specification and Description Language of the ITU-T, SDL. This compiler uses conventional techniques of a hidden representation of the abstract syntax tree and code generation (to C++).

Lately, we proposed an open framework for integrated tools for ITU-T languages that is provisionarily called ULF-Ware [2]. An overview of its architecture can be seen in fig. 1 on the following page.

Oversimplifying, ULF-Ware contains a model-based compiler for SDL. The input (in this case, a textual SDL specification) is parsed and a *model* in the SDL repository is generated from it that adheres to the *SDL metamodel*. The next step transforms this to a new model in the Java/C++ repository adhering

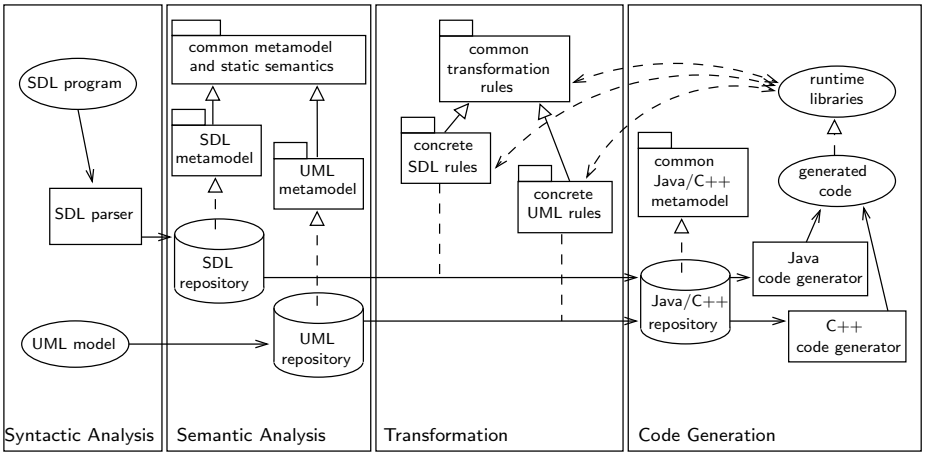

Fig. 1. ULF-Ware overview

to the *Java/C++ metamodel*. Finally, code generators turn this model into C++ or Java.

This paper shortly introduces the Java/C++ metamodel, a metamodel that is applicable to both Java and C++ and that comprises structural as well as behavioural aspects. The requirements for such a metamodel which is geared towards code generation are not obvious.

Similar metamodels exist, each with strengths and weaknesses. At the end of the next section, we will give a short overview of them. Mostly, they are concerned with structural aspects only. There is, however, a need for a new metamodel to cover all aspects of a programming language.

The second part of this paper is concerned with the runtime libraries. In SITE, we originally had only one output language (C++) and one library. Later, we added experimental support for Java [3]. In general, for n input languages and m output languages, you need $n \cdot m$ libraries and just as many different compilers/code generators. With the CeeJay metamodel as a common intermediate, this number should ideally decrease to just n transformations and m libraries.

Later, we added more C++ libraries that supported different needs: one library for simulation (including model time, stopping of the simulation, inspection of variables) and another for execution (only speed matters); also libraries that used different middle-ware platforms. These extra libraries did not require a new code generator, as they had the same interface as the old one. Generally, adding another library with the same interface does not require new transformations.

Therefore, this paper shows why a code generation metamodel and accompanying libraries are useful and simplify the task of translating SDL.

Section 2 will present general choices that had to be made in order to determine the shape of the metamodel in connection with the runtime library. Section 3 on page 5 presents the metamodel in detail. Finally, sec. 4 on page 7

describes runtime libraries as they are and how they will be interfacing with the CeeJay metamodel in order to simulate or run SDL systems.

2 Design considerations for CeeJay

The code-generation metamodel is conceived to be useful to generate C++ as well as Java from it. Therefore we have named it *CeeJay*.

2.1 The meta-metamodel

It is advantageous to employ the same meta-metamodel for all steps – this is a novel point of view: A conventional compiler might use BNF for the parser, but builds its abstract syntax tree using other means such as Kimwitu++ [4] or in an ad-hoc fashion.

We have chosen to use MOF as the meta-metamodel. It is used in many OMG standards, most prominently as the meta-metamodel for the UML. MOF is closely tied to UML, in fact there is a number of packages called the UML Infrastructure that are shared between MOF and UML.

MOF, however, is not than just a meta-metamodel, but it provides a meta-data management framework. There are a number of mappings from MOF to different platforms, such as a mapping to XML for model interchange and a mapping to Java which gives interfaces to create, navigate and modify models.

Using MOF and an appropriate tool for it gives a standard way to access models. First, you define a metamodel based on the MOF meta-metamodel. The tool then generates interfaces for models of this metamodel and an implementation to store the models. There are a number of tools, but the only one adhering to the new MOF 2.0 standard is "A MOF 2.0 for Java" [5].

2.2 Generic or specifc

High-level models are quite different from programs in conventional programming languages. They abstract from most of the detail that a programming language exhibits. Once you want to generate real code, all this detail has to be filled in. This makes code generation from those models a difficult task. Moreover, many decisions in this process are similar for different target languages, but it is hard to make use of these commonalities. Finally, the way back, i. e. from program text to high-level models, is very hard. Note that most tools that promise to do this (e. g. reverse engineering of Java to UML) only capture the structural aspects of the language (i. e. they only produce UML class diagrams).

The reverse approach is to use models that are very low-level and close to a specific language. There have been a number of papers such as [6] implementing this. The metamodel obtained this way is close to the original BNF of the language, they are grammars in disguise. Models like this are difficult to obtain. They would be the result of a model transformation from a high-level model. Here, the intelligence would have to lie in the transformations.

Thus the level of detail of the metamodel of a programming language determines whether there will be more work to do in the code generator or the model transformator. We have chosen a level of abstraction that allows true object-oriented models (as opposed to models closely relating to the syntax of a language) while still being close enough to programming to make code generation a straightforward process.

We will add another criterion to the decision as to how close to the target language the model should be: Can we use *one* metamodel for *many* languages?

2.3 Commonalities of object-oriented programming languages

Many languages share common concepts, such as the quite abstract concept of namespace. For programming languages, the similarities go even further.

Many differences in those languages are purely syntactical or for simple static semantics, such as the declaration of variables before use. The most important differences are support for crash-avoidance and the extent of the available libraries, neither of which affect the metamodel.

Java and C++ in particular are very similar to each other. Still, a complete metamodel would exhibit a number of fine differences such as the (non-)existence of multiple inheritance. However, we want to use Java and C++ as output languages only. This allows us to build a metamodel that can represent only the intersection of features from Java and C++.

Since Java and C++ have so much in common, the combined metamodel is still expressive enough to allow arbitrarily complex models. In fact, other object-oriented languages share the same concepts in very similar ways.

Some of the differences between languages are evened out because we want to use the models only for code generation. In Python, for instance, variables do not have a declared type. While generating code from a model containing type information, it is easy to just suppress generation of type names.

2.4 The Role of the Runtime Environment

The runtime environment serves two purposes: It hides differences between target languages and facilitates code generation for complex concepts.

While the target languages that we consider are semantically similar and exhibit mostly syntactical differences which are easily covered by the code generator, they have vastly differing standard libraries. The problem is already obvious in a simple Hello-World program. While C++ uses `printf` or `cout`, Java instead has `System.out.println`. How will this be represented in the model in a uniform fashion? One way is to have special metamodel-elements for printing text, and similarly for all the other library calls that differ; this also means changes to the metamodel if we want to include another call. The other way is to use a common runtime library that offers a uniform interface to the model and encapsulates differing functionality; clearly, this approach is superior.

In theory, code generation can generate code down to the most basic level. This would have the advantage of a minimal runtime library; this library would

be specific to the target language, but unconnected to the source language. But, as outlined in sec. 2.2 on page 3, it is preferable not to burden the code generator with too much detail. Instead, the runtime library contains support for the advanced concepts of the source language. While this results in different libraries for different source languages, it greatly simplifies the code generator. In the past, this approach was proven to be very successful in SITE.

The concepts of the library will be elaborated in sec. 4 on page 7.

2.5 Related Work

There are a number of metamodels for different languages around. However, the public availability of these metamodels is limited. Further, the focus of the metamodels can be quite different, as mentioned above.

There is a project called jnome [7]. The metamodel developed therein is tailored to generating documentation for Java files. It lacks support for the implementation of methods and is as such not suitable for complete code generation.

Both the Netbeans IDE and Eclipse seem to use a Java metamodel internally. Both IDEs bring their own repository functionality: Netbeans MDR, a MOF 1.4 repository, and Eclipse EMF, a repository and metamodel that is similar to MOF. Both metamodels are not MOF 2.0 metamodels and do not cover dynamic aspects.

The Object Management Group has a specification containing a Java metamodel [8]. which seems to have been abandoned in an early stage of development.

Numerous other works are concerned with automatically generating metamodels from grammars. The results are usually close to the grammar and, naturally, specific to the language they are based on. They are not suitable for a more general approach.

A metamodel that captures some common features of Java and Smalltalk is presented in [9]. This metamodel is concerned with common refactoring operations in those two languages.

3 The CeeJay Metamodel

This is a condensed overview of the metamodel. For a more elaborate description, please refer to [10].

3.1 Basic building blocks

The basis for the CeeJay metamodel is a number of classes that are not shown in a diagram here. They have self-explanatory names such as `NamedElement` and are modeled very closely to the elements of the same name as in the UML Infrastructure.

In fact, these classes can be viewed as a stripped-down version of the Infrastructure. It remains to be seen whether it is advantageous to have them here, which will aid in understanding the concepts because it is simple, or whether we should rather use the Infrastructure itself.

3.2 Packages

An important structural concept of Java is the *package*. In C++, there is the concept *namespace*. The two concepts are equivalent. The differences in usage are of no concern to the metamodel of the languages. While generating code, care is taken that every generated C++ namespace has an associated header file declaring everything, such that the inclusion of it is equivalent to the import of a package in Java; in Java, files are distributed into directories of the package's name.

Consequently, a single metaclass `Package` (a specialization of `Namespace`) suffices for both Java packages and C++ namespaces. They will be mapped accordingly, but see also 3.5 on the next page for an exception.

3.3 Class

Java does not allow multiple inheritance, so CeeJay can only support single inheritance for `CeeJayClass`. Java does, however, have interfaces. Those are `GeneralizableElement`s, and a `CeeJayClass` or an `Interface` can implement or enhance any number of `Interface`s.

While C++ does not have interfaces, they can be represented by abstract classes. Thus, the metamodel is applicable to C++ as well. A language without multiple inheritance and without interfaces may be difficult to map into from CeeJay models.

3.4 Types

The Types diagram in fig. 2 shows the different types in CeeJay. There are three main groups of types: Primitive types, classes and collections.

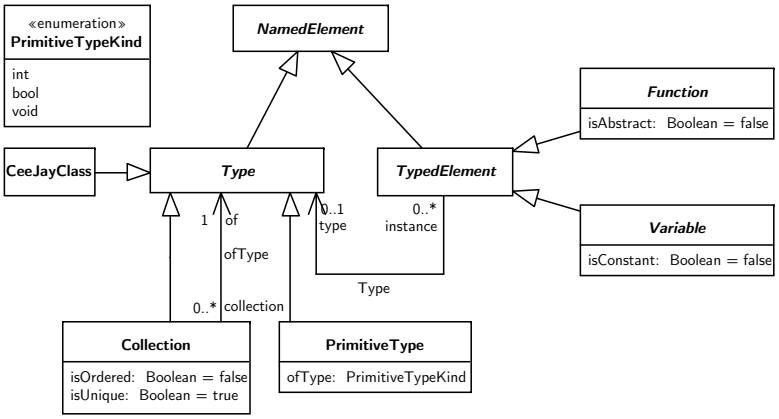

Fig. 2. Types, functions and variables diagram

Primitive types are the predefined types of Java and C++. As a start, only integers and booleans are used. The third primitive type here is `void`, which is used in functions (see 3.5 on the next page) as the return value.

There is a constraint on the primitive types that is not expressed formally: There may be only one instance of the metamodel element `PrimitiveType` for each `PrimitiveTypeKind`, and there always has to be exactly one. This means that there will have to be a number of model elements in CeeJay models that are always supposed to be there, just as the package `Predefined` in SDL.

The second variety of types is `CeeJayClass` itself. In both Java and C++, each class is also a type.

Collections are mentioned explicitly in this metamodel. In order to use the built-in collections of C++ or Java although they have different names (and different semantics and usage), there has to be a specific representation to unify them.

3.5 Functions and variables

As can be seen in fig. 2 on the facing page, both variables and functions are model elements that have a name and a type. Additionally, functions have parameters (ie. they inherit from `ParameterizedElement` and a body.

Variables can be marked as being constant. This, however, necessitates initial values, and those can imply an order for the variables, which is not desired.

Function bodies It is not yet completely clear how to represent the body – is it a nested namespace containing, among other things, the local variables, or is it only an ordered sequence of statements? The latter approach has the advantage of being simple, but it is also very close to an abstract syntax tree and as such not in the spirit of modelling. For instance, declarations of variables would just occur in the statement sequence, the variables would not truly live within the funtion's namespace.

The choice of the representation of the body is heavily influenced by the way the runtime library is built. While the languages that we want to generate code for are similar, their system libraries are very different. Parts of these differences will be covered by the metamodel, others by the supporting runtime library (cf. sec. 4.2).

The requirement that each function need a type leads to the need for a type `void`, which has to be explicitely disallowed for variables. The respective constraint is not shown here. The alternative solution, one that was also chosen in the UML metamodel, is to have the type being optional on a `TypedElement`, which, unfortunately, is not intuitive at all.

4 The Runtime Library

4.1 Related Work

Automated code generation has become an accepted method in the design of new software systems (protocols, IT switches, control systems). There is a wide range

of code generators available which produce code of a particular programming language from a particular specification or description language like SDL.

Most of them use some kind of runtime environment for the generated code for reasons of abstraction, efficiency and adaptability. The environments could be realized quite differently. Some SDL code generators use a code library (like Telelogic TAU [11]), some use preconfigured code fragments (like SINTEF ProgGen [12]), and others use runtime environments in a even broader sense. Usually each code generator is quite fixed regarding its runtime environment, target language, target platform and the configuration of the resulting executable system.

SITE [1], the SDL code generation environment once developed in our research group, behaves similarly. Its generated C++ code depends on a runtime library. But various efforts have been undertaken to provide more flexibility. It is possible to exchange the library when conforming to its interface, and there is a mechanism to change the code generator output without changing the generator, and additional information could be embedded directly into the SDL specification. We perceive the desire to be more flexible, to reuse the existing generator for a broader spectrum of application. The runtime library presented in [13] is meant as a contribution to that issue. That library allows the generated system to be adapted to different communication means to the environment, that is protocols and encodings.

But restrictions remain. If we want to switch to a different target programming language, even if close the the one used, a completely review of the code generator is needed, possibly even a new code generator. Inevitably, we need a new runtime library, too. That is, for every desired combination of source and target language we need one code generator. If we could achieve a separation of syntactical issues from abstract concepts of the target language it would be sufficient to have one abstract transformation per source language and one concrete code generation per target language.

4.2 SDL runtime library

Elements of the library One aim of our code generation is simple and readable code; another is a straightforward code generator. Both aims imply that the runtime library must give extensive support for the concepts of the source language. As has been previously shown in [1, 3] this can be achieved by structural equivalence of the source and the target specifications.

The result can be seen in fig. 3 on the next page. The system type `stype` is transformed into a C++ class of the same name that inherits from `sdlsystem`, a class defined in the library. Likewise, `btype` is a block type in SDL and inherits from `sdlblock` in C++. The gate and the channel are simply variables of the corresponding C++ types.

To this end, the runtime provides classes for SDL concepts that the concrete classes of the specification inherit. Other elements of the library not shown here are functions to route signals.

```
signal alive;
system type stype;
    gate whorl out with alive;
    channel pingpipe from b via bwhorl
        to env via whorl with alive;
    block type btype;
        ...
    endblock type;
    block b : btype;
endsystem type;
system s;
```

```
class alive : sdlsignal;
class stype : sdlsystem {
    sdlgate whorl(0, alive);
    sdlchannel pingpipe(b, whorl,
        env, whorl, alive);
    class btype : sdlblock {
        ...
    };
    btype b;
};
stype s;
```

Fig. 3. SDL and corresponding C++ (simplified)

Interfacing models and the library In order to use the library effectively, the library's interface has to be specified. Often, this happens in plain English text. However, to access the library in the model, we need a formal definition of the interface in a package that is part of the model, just as a set of header files for a C library become part of the program at compile time. In a similar fashion as the package `Predefined` in SDL, this package is considered to be in every CeeJay model. This package, however, does not contain any implementations, and no code will be generated for it; it is considered to exist in the target as well. This is necessary to use references in MOF (for specialization or function calls), as those are not by name.

Furthermore, the interface must be consistent for all the runtime libraries (for different requirements and target languages). In the past, this was often not enforced, and differences between the libraries became apparent only when problems occured.

To define the interface, we have yet to decide upon a standardized method such as IDL or eODL. Note that the interface is not limited to function calls, but also includes classes that are used as types and for inheritance.

5 Conclusion

The last step in a complete model driven software engineering process is the generation of implementation artifacts, usually in the form of source code. While code can be generated directly from high-level models such as UML, this puts too much work on the code generator. Instead, a stepwise refinement of the model into a model geared towards code generation is preferable.

To this end we have prepared a metamodel that is reasonably close to the target languages Java and C++, while still being general enough to not only cover these two languages, but other object-oriented languages as well.

This is different from existing metamodels that have been published (mainly for Java), which are close to the grammar of Java. Thus, they can often hardly be called metamodels, as they are no more than a MOF representation of the abstract grammar.

The CeeJay metamodel will be used in a framework where C++ is generated from SDL specifications. The aim of this open framework is to extend it for other

output languages, such as Java or Python, and other input languages, such as UML or domain specific languages.

In order to execute SDL models, infrastructure is needed that will be included in the form of runtime libraries. These facilitate the code generation by covering the remaining differences between our target languages and by providing solutions for complex concepts of the source language (SDL). Still more importantly, an infrastructure with a well-defined interface allows to change the runtime library to meet specific new needs.

The code-generation metamodel CeeJay and its accompanying libraries will allow us to generate code for and execute or simulate arbitrary SDL systems.

References

1. Schröder, R., Böhme, H., von Löwis, M.: SDL Integrated Tool Environment (1997–2003). URL `http://www.informatik.hu-berlin.de/SITE/`. Last checked: February 27, 2006
2. Fischer, J., Kunert, A., Piefel, M., Scheidgen, M.: ULF-Ware – an open framework for integrated tools for ITU-T languages. In: Prinz et al. [14], 1
3. Neumann, T.: Abbildung von SDL-2000 nach Java. Dissertation, Humboldt-Universität zu Berlin (2000)
4. Neumann, T., Piefel, M.: Kimwitu++ – A Term Processor (2002). URL `http://www.informatik.hu-berlin.de/~tneumann/manual.html`. Last checked: February 27, 2006
5. Scheidgen, M.: A MOF 2.0 for Java. URL `http://www.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava`. Last checked: February 8, 2006
6. Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Tucs technical report no 606, Turku Centre for Computer Science (2003)
7. Dockx, J., Mertens, K., Smeets, N., Steegmans, E.: jnome: A Java meta model in detail. Report cw 323, Department of Computer Science KULeuven (2001)
8. OMG: Metamodel and UML Profile for Java and EJB Specification. Object Management Group (2004). formal/04-02-02
9. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Proceedings ISPSE 2000. IEEE (2000), 157–167
10. Piefel, M.: A common metamodel for code generation. In: J. Aguilar (ed.), Proceedings of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications. IIIS, Orlando, USA (2006)
11. Telelogic: Communications software specification and software development Telelogic TAU SDL Suite (2006). URL `http://www.telelogic.com/corp/products/tau/sdl/index.cfm`. Last checked: February 28, 2006
12. Floch, J.: ProgGen: SDL transformation Tool. SINTEF Telecom and Informatics (1998). URL `http://www.sintef.no/units/informatics/products/proggen/`. Last checked: February 28, 2006
13. Fischer, J., Neumann, T., Olsen, A.: SDL code generation for open systems. In: Prinz et al. [14], 313
14. Prinz, A., Reed, R., Reed, J. (eds.): SDL 2005: Model Driven: 12th International SDL Forum, volume 3530 / 2005 of Lecture Notes in Computer Science. Springer-Verlag GmbH (2005). ISBN 3-540-26612-7. ISSN 0302-9743